University of Groningen

Rapid user interface development with the script language Gist

Bos, Gijsbert

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
1993

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Bos, G. (1993). *Rapid user interface development with the script language Gist*. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 05-05-2023

Chapter 3

User interface development systems

Programs that sport a graphic interface are typically very large and complex, because handling of the bitmapped screen and reacting to all different types of input involves a lot of work. To facilitate the work, sets of algorithms and whole libraries of (graphics & windows) subroutines are being developed (e.g., X Windows) that are often also commercially available (e.g., Microsoft Windows). In all these systems, however, it remains the case that the programmer has to work in a programming language that is primarily geared to working with simple objects like numbers and letters.

Languages that use *object-oriented* techniques are much better suited to handling the complex structures needed for graphic systems. In languages like Smalltalk[1] data and procedures are not considered separate. Instead one starts from *objects* that are defined by the data they can contain as well as the operations performed on them (section 3.1.3).

A different approach is to use a special-purpose language. Such a language is often part of a larger system, a UIMS or UIDE. Some UIDE's even forego the use of a language and let the designer assemble an interface by direct manipulation.

Of course, as soon as interfaces with windows, icons and a mouse become easy to make, the road is clear for new ideas that are waiting already, such as voice-I/O and devices that track the movement of your head or eyes. But even if interface-building tools serve no purpose other than speeding up the development of new ideas, that is enough reason to go ahead with it. Or as Edsger W. Dijkstra said it in his 1972 Turing Award Lecture: «[...] once we have freed ourselves from the circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity.»[2]

[1] see Goldberg [1983]

[2] see Dijkstra [1987]

3.1 Software development techniques

Most modern software engineering methods stress the importance of good, precise specification, before actual coding starts. But in the design of user interfaces this technique is not advocated. The method called *iterative design* – meaning design that is regularly adjusted after field tests – is much better here, the main reason being that people are complex and seldom certain about what they want.

3.1.1 Specification vs. iterative design

This presents a dilemma for software developers. How do you reconcile formal specification of one part of a system with the need for experimentation in another part?

The first step towards solving the problem is to be aware of it. Firstly, to take it into account when planning the development effort. Secondly, to be able to trace a problem back to the appropriate part of the design.

This seems to call for a separation of the software in two parts (a modular approach, or even a client-server model, see 1.6.7) and a definition of the interface between them. This solves most of the problems for many programs, in particular those that aim to solve a clear-cut problem. When the task can't be formalised so easily, the problem may require other techniques, because the development of the user interface may still call for changes in what the program does.

Two examples clarify the distinction. First look at a database engine that needs a user interface. The database has been specified precisely, complete with its own algebra. In principle, it can do anything a database needs to do. All that remains for the user interface is to present a particular view. You can choose a radically different interface without it having any consequence for the database.

As an example of a system that can't be specified completely beforehand, consider an electronic messaging system between people in a work group. Experimenting may reveal that users do not just send messages, but that they try to quote and refer to earlier messages, sometimes even several earlier messages. A system that was designed to have two modes of operation – reading mode and writing mode – may need redesign to allow reading and copying of messages even while composing a new one.

A number of techniques are available for assessing the quality of an interface. Depending on available time and resources,

the researcher might use questionaires, interviews, protocols, observation with video, performance testing, etcetera.

3.1.2 Prototyping

The ability to use graphics and a mouse (or any other pointer device) opens up a range of possibilities for interfaces. Should you associate a certain action with an icon, a menu or a button? Should the mouse be «clicked» once or twice? Does moving the mouse have any significance? Do you attach a meaning to the position of the mouse? Do you use colours? Large letters? Animations?

Little can be said in advance about how effective an interface will be. Much depends on the user's experience, the similarity to other programs and, of course, taste. The best way is often just to try and see: make a prototype and test it out, collect protocols of the way people interact with your program, evaluate their reactions and make a new prototype; and so on until you are satisfied.

However, it is not often that you have the time and facilities to actually test a prototype and discard it for something else. And after spending months implementing, who would be willing to do it all over again? The problem, again, is that general purpose programming languages are not very well suited and that even with precompiled libraries of useful routines, the programmer has to do a lot of things over and over again.

3.1.3 Object-oriented programming

In object-oriented programming a program is made up of objects that communicate. Each object represents some data, together with the operations that are defined for that data. The implementation of the operations is hidden inside the object (encapsulation). Each object is of a certain type, called the object's *class*. There may be general and more specific classes. A more specific class is usually a *sub-class* of another class, which means that the subclass *inherits* all operations from the *super-class* and adds some more. It can also add more dimensions to the data that is kept in the object. An operation in an object is activated by sending a *message* to the object.

The same message can mean different things to different classes of objects. This is called *polymorphism*. In contrast, a procedure in a procedural programming language has a unique function. It can only be varied by substituting different values for the parameters of the procedure. E.g., in a graphics program an *icon* can be an object: it contains information about

the appearance of the picture and information about possible changes (different colours, different position on the screen, et cetera).

This method of software design better matches the picture that the user (and the designer!) has of the final interface: a collection of objects on the screen, each of which he can manipulate independently. Smalltalk and the X Toolkit library are examples of object-oriented systems.

3.1.4 Lazy & eager evaluation

There is a difference in the way processing proceeds in the application and interface parts of a program. Most programs are written in *imperative* languages, that support only *eager evaluation*. In this type of program every part of an expression and every argument to a procedure is evaluated, even if later it proves to be unnecessary.[3]

On the other hand, the event-driven processing required by graphic interfaces is more like the *lazy evaluation* type of processing, where computation is postponed until absolutely necessary. This also means that inputs are not processed in the order dictated by the program, but only as soon as and in the order as chosen by the user. The system must be able to work with and display partial results.

3.1.5 Non-deterministic design

An interesting reversal of the role of user interface design is mentioned by Thimbleby [1990] (in box 8.4 on page 166 of his book). *Non-deterministic design* is design for which fitting users will be found afterwards. It isn't that uncommon and it isn't necessarily bad.

It is like an economic market with many producers and many consumers. Many will find a satisfactory partner, because of all the variations.

3.2 Some history

Even in the fifties there were some graphics terminals available, mostly for the display of graphs and other plots. They often had a limited form of graphical input with two thumb wheels that moved a hairline cross on the screen. A more ambitious use of graphics was Ivan Sutherland's Sketchpad program of 1962. It allowed geometric forms to be drawn on the screen, and the corresponding datastructures to be maintained internally.

The use of graphic interfaces got a boost at the end of the seventies with the development of three computers; Xerox

[3] The exception is usually the evaluation of Boolean conditions. In the expression

`if` $A \wedge B$ `then...`

B need not be evaluated when A fails. A and B can be quite complex, even involving user interaction. Lazy evaluation in this case saves the program — and possibly the user — from unnecessarily processing B. Ideally, if A and B both involve actions on the user's part, the user should have the option of answering B before A.

Alto, Lilith and Xerox Star. These computers used a screen that could display high resolution graphics and they received input from an input device that was called a mouse, that was invented at Doug Engelbart's Augmentation Research Center at Stanford, some years earlier. Overlapping windows were also introduced on these machines as was the desktop metaphor. At Xerox Palo Alto Research Center (PARC) the Xerox Alto computer was developed in 1974 for internal use. It was the sort of machine the developers wanted for themselves. It had 64Kb memory – for that time a large and expensive amount – and a black and white graphics screen with about half a million pixels (comparable to modern super-VGA).

The Lilith computer was created in 1977 by a team headed by Niklaus Wirth in Zürich, after Wirth spent a year in the laboratory of Rank Xerox in Palo Alto, where he helped develop the ideas that led to the Xerox Star computer and the first version of the programming language Smalltalk (1976). The Star was the only machine that was sold commercially (in 1981), but it was very expensive and few people could see the benefit of the new system.[4]

The young computer company Apple also tried to turn the Xerox ideas into a commercial product, but their attempt, the Lisa computer, was no success. The value of the ideas was recognized, however, by Alan Kay, one of the major forces behind the Alto, who left Xerox in 1980 and some years later joined Apple to help design the Macintosh computer (1984), the machine that introduced millions of people to GUI's and made them popular. (The hectic years when the Macintosh was born are described in «West of Eden».[5]) In 1985 already other manufacturers followed Apple's example and created machines or software or both for graphic interfaces (Commodore Amiga, Atari ST, Digital Research's GEM for Atari and MS-DOS, Microsoft Windows).

The next notable step in this chronology was again made by Apple. In 1987 the Macintosh was enriched with a piece of software called *HyperCard*. This is a system that allows users to design simple programs that make use of the capabilities of the Macintosh. It is in a sense a replacement for the programming language BASIC, that was often bundled with earlier computers for exactly the same reason.

HyperCard introduced a new metaphor, that of stacks of cards. Each card contains some information and some active elements, such as links to other cards or programs to execute. The user can go forward and backward through the cards or

[4] see Degano and Sandewall [1983] for a description of the Alto.

[5] see Rose [1989]

go to another stack. Predefined elements can be combined with texts and images to create, for example, a database stack or a game stack. When the predefined elements are inadequate, there is a programming language, HyperTalk, with which small programs can be written that are attached to buttons, stacks, icons, etcetera.

HyperCard and HyperTalk are especially important, because they are explicitly meant to be used by end-users, not programmers. The HyperTalk language is kept simple and readable.

The Trillium user interface design environment created at Xerox PARC in 1983 was one of the first systems specifically for use by the designers of interfaces. In this case the goal was to simulate the interfaces of various machines, such as copiers and printers. Fast prototyping was the main requirement. It is an interpreter – the «Trillium machine» – that manages an interface built up from various types of objects or «frames», that react to changes in their environment, such as the press of a mouse button. The actions in Trillium are coupled to so-called *sensors*. Henderson [1986] describes the Trillium system in more detail.

The X Window System is a combination of a window system and a network system. It uses a client-server model. The server manages a display and displays whatever the clients request. There may be any number of clients, running on any number of machines. The clients own one or more windows. To ease programming, X provides a library of routines for common operations, such as opening and closing windows, drawing lines, writing text, etcetera. Even with this «Xlib» library, programming is very complex, and additional, higher level libraries are provided on top of Xlib, such as the X Toolkit.

X Windows was finally distributed in 1987. It had been growing from 1984 through 10 versions at the MIT (version 10 was released in 1986). The introduction of the X Window System meant that now nearly all UNIX workstations could use the same window system, which made them much more popular.

In 1988 the NeXT computer was introduced: a UNIX workstation with built-in support for sound and digital image processing and also an object-oriented tool for designing interfaces. The tool, NeXT Interface Builder, makes it easier for programmers to create GUI's on this computer.

In 1989 a new portable computer was introduced, the GRiDPad, a so-called pen-computer. It is a computer without a

keyboard, but with a stylus with which you can write directly on the screen. The computer can recognize the handwriting. Since then a number of similar computers have appeared, but the technology, especially the handwriting recognition software, is not yet mature enough for widespread use.

HyperCard had a lot of followers, both on the Macintosh and on the IBM PC under Microsoft Windows. A different approach is taken by the program Matrix Layout, which provides its own graphic routines on the PC and uses them to let users create programs by drawing flowcharts on the screen.

The latest contribution from Xerox is the *Information Visualizer,* a way of representing information on screen that uses perspective drawing and animation to pack more data in a small space. The metaphor is no longer the flat desktop with documents and tools scattered all over it, but a collection of *rooms* with walls, doors, and a floor, containing one or more objects. Perspective drawing allows the interface to show more parts of an object (a directory tree, for instance) by making them smaller. To have a closer look at another part of the structure, you rotate it smoothly until the required part comes into view. Zooming in or out is likewise animated. The perspective and the animation give you additional ways of seeing the relations between parts of an object and these clues are presumably processed unconsciously – and therefore very fast – by parts of your visual system.

It is interesting that the Information Visualizer uses a normal keyboard and mouse and no additional hardware. But the fact that the current – experimental – implementation only runs on a Silicon Graphic Iris graphics computer suggests that the system isn't yet ready for the average workstation (or the other way round: that workstations still need to become more powerful).[6]

Meanwhile, the X Window System has spawned many new developments. The X Toolkit defines *widgets,* which are complete, self-contained implementations of interface elements, contained in libraries that can be linked to a program. The widgets allow programming in a more or less object-oriented way. Various vendors have provided sets of widgets to ease and illustrate programming in their preferred style. Examples are the Motif widget set and the Open Look widget set. There are also many public domain widgets and widget sets, such as that of the Free Widget Foundation.

There are also programs marketed by hardware manufacturers to make programming for X on their machines easier (Hewlett Packard's Interface Architect, e.g.). And there

[6] see Clarkson [1991]

are public domain programs like David E. Smyth's Wcl and Richard Hesketh's Dirt, that try to make widget programming more flexible.

3.3 Current systems

A number of UIMS's and UIDE's is currently available, either commercially, free, or for research purposes. An up-to-date list can be found in Heeman [1992]. UIDE's use different approaches, which can be broadly classified as toolkits, interactive GUI builders and script-based systems. (See also figure 3.1.)

A toolkit is nothing more than a library of routines, that can be called from a program. Toolkits may provide drawing routines, windowing routines, and input routines. Examples are the standard X Toolkit, InterViews, CommonView, MacApp and XVT. **InterViews** has been created at Stanford University.[7] It consists of a set of C++ classes. The most notable feature is the way in which screen layout is specified: it uses the boxes-and-glue paradigm of TEX. **CommonView** is commercial product by Glockenspiel. There are versions for Presentation Manager, MS Windows and X, which means that (almost) the same C++ programs can be compiled under all three window systems. **MacApp** by Apple[8] is a library for Object Pascal and C++ on the Macintosh. **XVT**[9] is a library for use with C. Like CommonView, it offers versions on a number of platforms, viz. X11 with Motif, X11 with Open Look, Macintosh, MS Windows, Presentation Manager and text-mode under DOS and Unix.

Interactive GUI builders are programs that let a programmer create an interface by direct manipulation, i.e., by dragging and dropping interface objects, until the screen looks like the desired layout. Unfortunately, this method can only be used for specifying the appearance of the interface, the dynamics must still be programmed, unless the defaults are good enough. Some examples are Dirt, Druid, NeXT Interface Builder and Visual Basic.

Dirt[10] is in the public domain. Dirt runs under X and outputs a resource file that must be read with Wcl. The behaviour of the interface cannot be specified graphically, but must be programmed in the form of callback routines. **Druid**[11] generates C and Motif UIL (User Interface Language). Part of the dynamics can be specified interactively as well. The **NeXT Interface Builder** «IB» outputs Objective C code. Some standardized dynamics can be entered graphically. **Visual Basic**

[7] see Linton, Vlissides and Calder [1989]
[8] see Shmucker [1986]
[9] see Rochkind [1989]
[10] see Hesketh [1992]
[11] see Singh, Kok and Ngan [1990]

	Research	Commercial
Toolkits	ET++ *Interviews PCE *SUIT WINTERP *X Toolkit	*CommonView GUI_Master *MacApp OI (Object Interface) *XVT
Interactive systems	*Dirt *Druid Fabrik *FormsVBT Garnet Ibuild MoDE Peridot Programming by Rehearsal Serpent TAE Plus UIDE *DIGIS	Builder Xsessory DataViews DEC VUIT Devguide ezX Interface Architect LAF Toolkit *NeXT Interface Builder TeleUSE UIMX ViewEdit *Visual Basic XBuild *Matrix Layout *X-Designer XFaceMaker2
Script-based	HyperNeWS *Tooltool *WCL *Motif UIL *Tcl/Tk *Gist	*HyperCard/HyperTalk NewWave Prograph Serius89 IBM dialog manager

Figure 3.1 *A table of user interface development systems. The table is an extended version of table 2 from Heeman [1992]. The products that are marked with an asterisk (*) are mentioned in the text of this chapter. Gist is the author's own system, described further on in this book. The «script» section contains both non-interactive script-based systems and interactive script-based systems. Additional (commercial) systems can be found in Peddie [1992].*

runs under MS Windows. It uses the programming language Basic for programming the dynamics. **X-Designer**[12] also provides a direct manipulation editor for laying out an interface. It generates C, C++ or Motif UIL. The C or C++ functions that it creates are empty stubs, that must be filled out with additional C code. For that reason, X-Designer can be combined with a CASE tool, called CodeCenter. It is targeted at programmers only. It is difficult to change an interface, since

[12] see Bergmans [1993]

regenerating the C stubs causes any code that was put into them to be lost. Changing the layout or the default resources is no problem, however.

Script-based systems read a description of the interface and either generate a program or directly interpret the script. A «script language» is a special purpose programming language, in this case for defining interfaces. Like all programming languages, it can be either declarative or procedural. Procedural languages make use of algorithms, declarative languages define relations and constraints. Procedural languages are not very well suited for non-programmers, but they may give programmers better control. Most script must be typed in first and are then executed, but some systems allow the script to be changed on the fly.

The Motif UIL is an example of an interpreted language, as are Wcl[13] and HyperCard.[14] **Matrix Layout** is a sort of halfway system between a script and a direct manipulation system. It builds a program by assembling a flowchart interactively. The flowchart is then translated into either an executable program or source code for Pascal, Basic or C. The program can also be executed and previewed directly from the editor. It runs under DOS and MS Windows.

A previous version of **FormsVBT**[15] allowed the designer to switch between a graphic direct manipulation editor and a text editor, where a textual representation of the interface could be edited. In the newest version (July 1992)[16] the direct manipulation editor was dropped and only the editor for the Lisp-like language was kept. It is still possible to get a preview of how the interface will look, by pressing a button in the editor. FormsVBT can only be used with Modula-3 (formerly Modula-2+) programs, which must be specifically written for use with one of the VBT libraries. Figure 3.2 shows a sample interface description in FormsVBT.

Tooltool is a UIMS that runs as a separate process in parallel with the application. The application runs as though it reads input from the terminal and writes output to the screen, but Tooltool intercepts the I/O and passes it through a window. It uses a script only for transforming user input, output is simply displayed in a text window. Tooltool has extensive support for expressions, almost as expressive as C. The script syntax is reminiscent of C in other respects, too. The types of interfaces that can be created with it are limited, however,

[13] see Smyth [1991]
[14] see Apple Computer Inc. [1988]
[15] see Avrahami, Brooks and Brown [1989]
[16] see Brown and Meehan [1992]

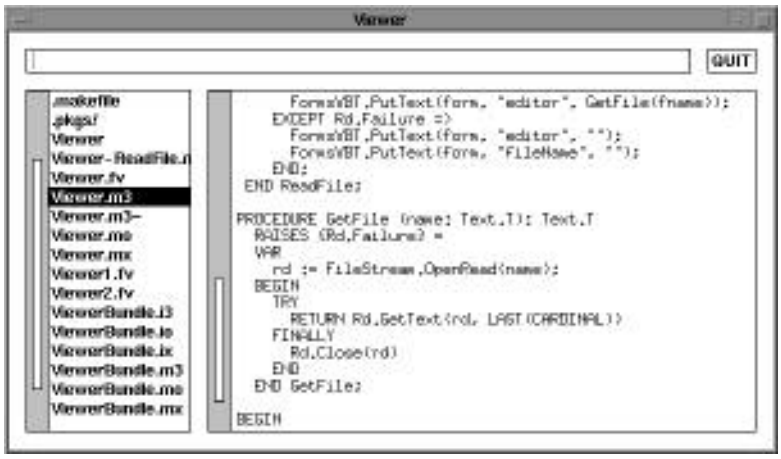

Figure 3.3 The file viewer corresponding to figure 3.2 in use. (This image and the text of figure 3.2 are taken from Brown and Meehan [1992], page 24)

since the number of interface elements is quite small and non-extensible, and since there can be no hierarchy among elements. Tooltool can create a main window and any number of dialog boxes, but all elements in a window or box must are placed directly on the window, they cannot be nested. Tooltool only runs on Suns under Sunview.

3.4 Advantages & disadvantages

Each of the systems mentioned has its own strong and weak points, but they cannot simply be compared. They target dif-

```
(Rim (Pen 10) (ShadowSize -1)
     (BgColor "White") (LightShadow "Black")
     (DarkShadow "Black")
   (VBox
     (HBox
       (Frame Lowered (Typein %fileNameString))
       (Glue 10)
       (Button %exit "QUIT"))
     (Glue 10)
     (HBox
       (Shape (Width 100)
         (Frame Lowered (FileBrowser %fileName)))
       (Glue 10)
       (Shape (Height 200 + inf) (Width 300 + inf)
         (Frame Lowered (TextEdit ReadOnly %editor))))))
```

Figure 3.2 An interface description in FormsVBT. Note the use of the Lisp-like S-expressions and the boxes-and-glue metaphor from TEX. The S-expression defines the interface to a simple file viewer. Each object in the interface must be bound to a procedure with the help of a library routine in the Modula-3 application program.

ferent types of users and claim different parts of the user interface design process. To make a broad categorization, we can distinguish between systems for programmers, for designers and for end-users.

(Professional) programmers can, of course, work with any of these systems. What differentiates them from the other two groups is that they are familiar with the concept of programming, that they understand about efficiency and that they are the designated people when it comes to forcing the last bit of performance from a system. They view an interface as just another program, defined by its datastructures and algorithms, of which only a projection is visible on screen.

By designers are meant those people whose primary expertise lies in the field of human-computer interaction. They may or may not be programmers, but for the purpose of this argument we assume they are not. Section 2.10 describes how they work. They approach user interfaces from the «outside»: the interface consists of the things that can be distinguished on the screen. The relations among them are usually not described in terms of hidden objects or mechanisms, as programmers would do.

When end-users are mentioned as creators of interfaces, the interfaces involved are usually small, single-purpose, and often temporary – the modern day equivalent of batch files or shell scripts.

End-users can also modify existing interfaces. Support for such *configuration* of an interface by the user can be built into the interface, or it can be provided by the same UIMS that the original designer used.

3.4.1 TOOLKITS

Toolkits are clearly targeted at programmers. Usually they are tied to a specific programming language, such as C, C++, Lisp or Prolog. They should be evaluated with respect to their fitness for use in programming. For this reason, and because the use of toolkits presupposes that the application and the interface are developed together, toolkits will not be dealt with in this text.

In fact, toolkits are often used in the creation of other systems. For example, many of the non-toolkit systems have been implemented with the help of the X Toolkit, including my own system, Gist.

3.4.2 Interactive systems

Interfaces have both visual and dynamic – or behavioural – aspects. Visual aspects include size, location, colour, decorations, etcetera. Dynamic aspects include constraints on what actions are available at what time and what action corresponds to which function of the application. Since interfaces are often designed by trial and error, it is seems logical to allow the interface to be created interactively. Many UIDE's employ some form of interactive design, often just for the visual aspects, but sometimes also to specify part of the behaviour.

Interactive systems usually employ direct manipulation to create at least the visual lay-out of the interface. Interface elements are dragged and dropped to their position on the screen, and they can be resized and static texts can be added to them. The relations among interface elements are much harder to specify with direct manipulation. Geometric relations can sometimes be added with the help of metaphors such as blobs of glue (FromsVBT) or springs (DIGIS).[17]

Various systems have also tried to provide some means of adding dynamic behaviour by direct manipulation. E.g. DIGIS allows a diagram to be drawn to specify in what order the various elements of an interface are activated. Interface elements can be collected into panes and then the same kind of diagrams can be created for the relations among panes, thus allowing hierarchical relations. Another system that tries to replace traditional programming with visual programming, although not strictly direct manipulation, is Matrix Layout. It uses flowcharts to specify procedures. The elements of the flowcharts can be selected from menus and inserted at the right place.

Dirt allows functions («callbacks») to be attached to interface objects. The functions have to be created beforehand and must be explicitly exported by the application. In DIGIS, the application must first be described with an object-oriented «Domain Application Model». The objects in the interface can then be linked to objects in the application model by means of various kinds of links or «signals».

3.4.3 Interactive script-based systems

Script languages have a number of advantages over both general-purpose programming languages and DM systems. Script languages are usually much simpler than general-purpose languages. They are easier to learn and they use

[17] see Van den Bos and Laffra [1990] and De Bruin, Bouwman and Van den Bos [1993]

high-level concepts that are close to the concepts in the task domain. And although the use of such languages seems a step backwards from programming by direct manipulation, this is not necessarily the case. Compared to DM systems, they have the advantages of being printable and easily copied in whole or in part. Also, a well designed language can often better express the actions of an object or the interface as a whole than the geometric relations of which direct manipulation must of necessity employ itself. At the very least programming with such languages is much faster than laboriously moving objects, on all but the smallest interfaces.

When the script can be edited interactively – i.e., with changes taking effect immediately, instead of after restarting the system – or when visual aspects of the interface can be modified interactively, the systems are still called «interactive». HyperTalk and Gist fall in this catagory. The combination of HyperCard with the HyperTalk script language seems especially popular.

With script languages the challenge is always to find a good compromise between the flexibility of a full-blown programming language and the learnability of much simpler systems. The designers of the HyperTalk language have tried to make a language that reads almost like plain English. There are no type declarations, but otherwise they have not tried to hide the control structures. Figure 3.4 shows an example. Chapter 4 describes how the dilemma has been handled in Gist.

Figure 3.4 An example of a HyperCard event handler. This script is attached to a card and asks a password when the card is opened. Note the use of the word «it» to refer to the most recently mentioned variable, in this case the result of «ask password».

```
on openCard
  repeat 3 -- the user gets 3 tries
    ask password "give password"
    if it is 11801 -- whatever is
                   -- the coded password
    then
      pass openCard
      exit openCard
    end if
  end repeat
  beep 6
  answer "Entry denied"
  domenu "Stop HyperCard"
end openCard
```

3.4.4 Non-interactive script-based systems

Some interface builders rely solely on scripts and do not allow interactive editing at all. Examples are Wcl and Motif UIL. Wcl extends the concept of resource as it is already present in the X Window System to include the possibility to specify a widget hierarchy and attach functions (callbacks) to other widget. The application must be specially written for use with Wcl, but the interface can then be modified considerably without changing the application.

Motif UIL has a syntax that is reminiscent of C, but the description is purely static. This is strange, since the similarity suggests that Motif UIL can be used for writing algorithms, but this is not the case. No dynamics can be specified, except for the attachment of callback functions that are defined in the application. The description is compiled and linked with the application, which means that a change in interface requires a recompilation of the entire application.

Tcl/Tk[18] uses the traditional UNIX shell as its model. Interface elements are created with commands that have arguments and options. In contrast to normal commands, the commands that create interface elements do not complete before the next command is started. Instead they leave an object on the screen. Figure 3.5 shows a Tcl script, the result is shown in figure 3.7. For comparison, the same interface is also specified in Gist, see figure 3.8. Clearly, Tcl is targeted at people familiar with imperative programming.

[18] see Ousterhout [1993]

```
# mkPuzzle w
#
# Create a top-level window containing a 15-puzzle game.
#
# Arguments:
#    w -        Name to use for new top-level window.

proc mkPuzzle {{w .p1}} {
    catch {destroy $w}
    toplevel $w
    dpos $w
    wm title $w "15-Puzzle Demonstration"
    wm iconname $w "15-Puzzle"
    message $w.msg -font -Adobe-times-medium-r-normal--*-180* -aspect 300 \
            -text "A 15-puzzle appears below as a collection of buttons.
Click on any of the pieces next to the space, and that piece will slide
over the space.  Continue this until the pieces are arranged in numerical
order from upper-left to lower-right.  Click the \"OK\" button when you've
finished playing."
    set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
    global xpos ypos
    frame $w.frame -geometry 120x120 -borderwidth 2 -relief sunken \
        -bg Bisque3

    for {set i 0} {$i < 15} {set i [expr $i+1]} {
        set num [lindex $order $i]
        set xpos($num) [expr ($i%4)*.25]
        set ypos($num) [expr ($i/4)*.25]
        button $w.frame.$num -relief raised -text $num \
                -command "puzzle.switch $w $num"
        place $w.frame.$num -relx $xpos($num) -rely $ypos($num) \
            -relwidth .25 -relheight .25
    }
    set xpos(space) .75
    set ypos(space) .75

    button $w.ok -text OK -command "destroy $w"

    pack append $w $w.msg {top fill} $w.frame {top expand padx 10 pady 10} \
            $w.ok {bottom fill}
}
```

(continued)

Figure 3.5 *A Tcl/Tk script for the 15-puzzle (see figure 3.7). Compare this script to the one in figure 3.8. (The text after «message» is in reality all on one line.) The Tcl script is clearly procedural, whereas Gist is declarative. Objects get their attributes via slot & filler (or keyword-value) pairs.*

(Continued from 3.5)

```
# Procedure invoked by buttons in the puzzle to resize the puzzle entries:

proc puzzle.switch {w num} {
    global xpos ypos
    if {(($ypos($num) >= ($ypos(space) - .01))
            && ($ypos($num) <= ($ypos(space) + .01))
            && ($xpos($num) >= ($xpos(space) - .26))
            && ($xpos($num) <= ($xpos(space) + .26)))
            || (($xpos($num) >= ($xpos(space) - .01))
            && ($xpos($num) <= ($xpos(space) + .01))
            && ($ypos($num) >= ($ypos(space) - .26))
            && ($ypos($num) <= ($ypos(space) + .26)))} {
        set tmp $xpos(space)
        set xpos(space) $xpos($num)
        set xpos($num) $tmp
        set tmp $ypos(space)
        set ypos(space) $ypos($num)
        set ypos($num) $tmp
        place $w.frame.$num -relx $xpos($num) -rely $ypos($num)
    }
}
```

Figure 3.6

Figure 3.7 *The 15-puzzle as it appears on screen when the scripts of figures 3.5 or 3.8 are excuted.*

```
#!gist

highlight-thickness 0                    # global default

/set (.*) (.*)/:                         # input from application:
    $1$location $2.                      #  set location of $1 to $2
/create (.*) (.*)/:                      # input from application:
    CLONE btn AS $1                      #  create button $1
    $1$location $2                       #  set location to $2
    $1$label $2                          #  set label to $1
    OPEN $1.                             #  open $1

OBJECT window "15-Puzzle Demonstration"
width 405
height 280

OBJECT label message
font "-Adobe-times-medium-r-normal--*-180*"
alignment left
margin 7
shrink-to-fit yes
label "A 15-puzzle appears below as a collection of\n\
buttons.  Click on any of the pieces next to the space,\n\
and that piece will slide over the space.  Continue\n\
this until the pieces are arranged in numerical order\n\
from upper-left to lower-right.  Click the \"OK\"\n\
button when you've finished playing."    # label doesn't do wrapping...

OBJECT board frame                       # this holds the 15 buttons
location "0.5-60 1.0-150 120 120"
background Bisque3
frame-type sunken
frame-width 2

OBJECT button btn (frame) CLOSED         # prototype for button
MOUSE-CLICK: PRINT SELF$label+" "+SELF$location+"\n".

OBJECT button OK                         # button along the bottom edge
location "0 1.0-25 1.0 25"
label OK
MOUSE-CLICK: HALT.
```

Figure 3.8 *This Gist script produces exactly the same interface as the Tcl script in figure 3.5, except that it doesn't contain the puzzle logic. A simple program like the one in Awk in figure 3.9 better describes the puzzle than it could be done in Gist. The puzzle is started with the command:* `puzzle.g puzzle.awk`.
Normally, the location of objects is a matter for the interface. But in this particular program the locations of the 15 buttons are under the control of the application program. The interface in this case only provides a prototype button.

```
#!/usr/bin/awk -f
# receives messages of the form "label x y w h"
BEGIN {
  order = "3 1 6 2 5 7 15 13 4 11 8 9 14 10 12"
  split(order, ord)
  for (i = 0; i < 15; i++)                          # create buttons
    print "create", ord[i+1], (i%4)*.25, int(i/4)*.25, .25, .25
  freeX = 0.75                                      # position of free square
  freeY = 0.75
}
{
  if ($3 == freeY && $2 + 0.25 == freeX) {          # move right
    freeX = $2
    print "set", $1, $2 + 0.25, $3, $4, $5
  } else if ($3 == freeY && $2 - 0.25 == freeX) {   # move left
    freeX = $2
    print "set", $1, $2 - 0.25, $3, $4, $5
  } else if ($2 == freeX && $3 + 0.25 == freeY) {   # move down
    freeY = $3
    print "set", $1, $2, $3 + 0.25, $4, $5
  } else if ($2 == freeX && $3 - 0.25 == freeY) {   # move up
    freeY = $3
    print "set", $1, $2, $3 - 0.25, $4, $5
  }                                                 # else, can't move
}
```

Figure 3.9 *The Gist script in figure 3.8 doesn't contain the puzzle logic. A simple program like this one in Awk better describes the puzzle. The program starts by generating the locations for the 15 buttons and sending 15* `create...` *messages to the interface.*